Specification of Make files for Modules and Applications

Version 2.0

1. Introduction 3
2. Structure 3
    a. Makerules: 3
    b. Interfaces: 4
3. Specification 5
    a. Make variables 5
    b. Identifiers 9
    c. Build options 11
4. How these make files work? 13
    a. Building modules 13
    b. Building applications 13
5. Appendix 15

1. Introduction

This document outlines the specification of make build system. This is meant for module/component developers and integrators of these modules. The document gives details of module and application make files.

2. Structure

The following figure gives a high level view of the structure of the make files.

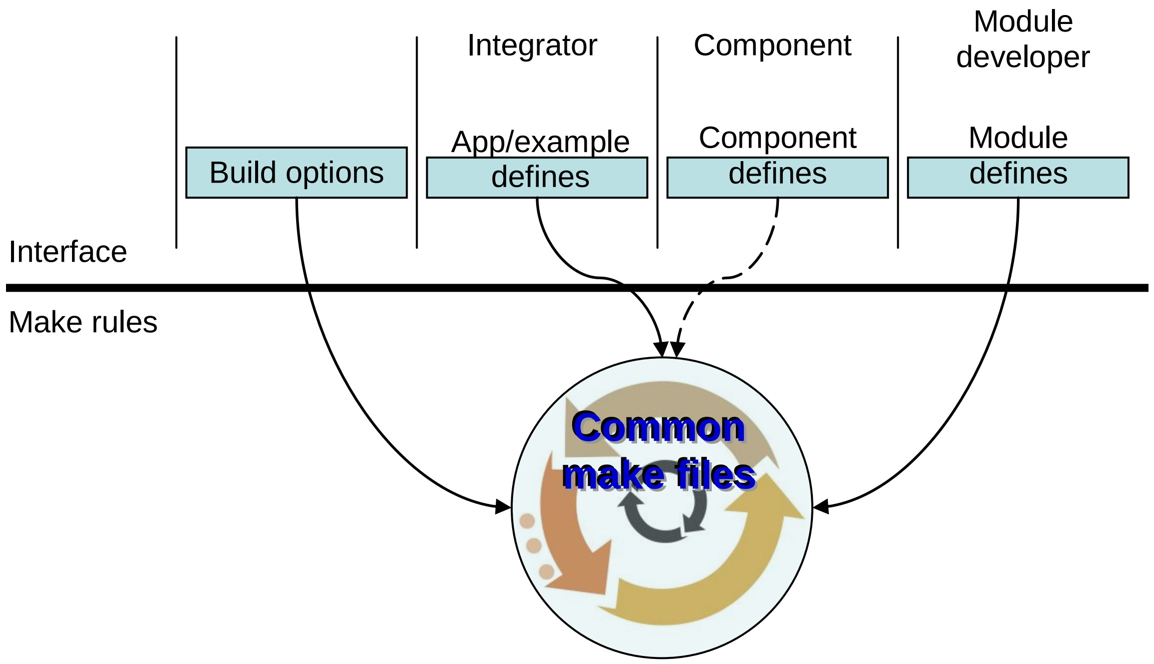

Figure 2.1

The make rules are implemented in a common set of make files that are used by module/component developers, middle-ware developers, sub-system integrators and high level app developers/integrators.

a. Makerules:

The complex part of make files is usually the rules – targets, dependencies and recipes. These are created as a template and shared and re-used across all levels of software stack. For example, rules to compile, archive and link for each of the ISAs are implemented as a part of these common make files. Other common build steps that are typically required to build TI’s software, eg. XDC specific build steps, are also a part of these common make files.

b. Interfaces:

- These common make rules are "configured" to work in a specific way. These would come from the "build options" that are defined by those who run the build. The option could be, for example, to build the complete stack for "little" endian-ness or build for ELF binary format, etc.

- Going from the highest level of the software stack, the integrators would integrate components and modules, add system/sub-system application/framework source code and would build an application or sub-system. They would specify app/example defines in the application's makefile to specify things like list of components/modules that are needed by the application, etc.

- In the next level, we have components. The definition of a component here is, it is just a collection of modules. Hence, in the strict sense it does not "use" common make files. The component's makefile would call other makefiles – that of modules and examples/apps that consume the modules. It would define several characteristics of the component and its modules. For example, relative path of each of the modules from the packages repository are defined. This makefile would also iterate the build process for multiple cores or boards, etc. as needed.

- The lowest level is a module. The makefiles of the modules specify the source files that would need to be compiled and archived in module's library, location of source files and the external interfaces (header files) that it includes, etc.

The defines in all these above levels of software stack are used by the common make files to build the required targets.

3. Specification

This section describes the specification of make files. Name of identifiers and variable names are specified here.

a. Make variables

The following table lists all the make variables that are set from either application makefiles or module makefiles. These variables are used by the common make files. It is important to maintain the case of the variables.

Variables in both application and module makefiles		
Sl	**Make variable**	**Description**
1	SRCDIR	List of directories (each space separated) where the source files are located. These directories would be relative to the directory where “makefile” for the app/module is located (top-level directory of the app/module).
2	INCDIR	List of directories (each space separated) where the local header files are located. These directories would be relative to the directory where “makefile” for the app/module is located (top-level directory of the app/module).
3	INCLUDE_EXTERNAL_INTERFACES	List of names of components whose interface header files the app/module includes in the source code
4	INCLUDE_INTERNAL_INTERFACES	List of names of modules whose interface header files the app/module includes in the source code
5	SRCS_COMMON	List of all C files (each space separated) which are common across boards, cores, ISAs and SoCs.
6	SRCS_<identifier>	List of C files (each space separated) which are specific to the <identifier>. Here, <identifier> could be: **<core>** eg: SRCS_ a15_0 lists C source files that apply to and are

		specific only for a15_0 core **<BOARD>** eg: SRCS_tda2xx-evm lists C source files that are specific only for tda2xx-evm **<SOC>** eg: SRCS_tda2xx lists C source files that are specific only for tda2xx SOC NOTE: List of all core names, board names, etc are listed later in the document
7	CFLAGS_LOCAL_COMMON	List of CFLAGS specific to the component/module. These flags will be passed as argument at the time of compilation

Table 3.1

The following table lists the variables that are required to be set by the module makefile (other than what is mentioned in *Table 3.1*).

Variables required to be defined by module makefiles		
Sl	**Make variable**	**Description**
1	MODULE_NAME	A unique single word that identifies the module. This name is used everywhere else in the make build system to identify the module.

Table 3.2

The following table lists the variables that are required to be set by the application/example makefile (other than what is mentioned in *Table 3.1*).

Variables required to be defined by application makefiles		
Sl	**Make variable**	**Description**
1	APP_NAME	A unique single word that identifies the application/example. This name is used everywhere else in the make build system to identify the application
2	COMP_LIST_< identifier >	List of name of components/modules (each space separated) that are required by the application to build the executable for a specfic <core> or “COMMON” across all cores. Each <core> used by the application should

		have this entry separately.
3	XDC_CFG_FILE_<core>	If the application uses XDC packages//components, then this has the file name of the configuration file name of the CFG file (this is parsed and used by the XDC's configuro command.
5	XDC_CONFIGURO	Flag to mention whether XDC configuro needs to be performed for this application. Set to yes/no. This is used to have both XDC and baremetal application to use the same make infrastructrue
4	EXTLIB_LIST_<core>	List of names of components/modules whose libraries have to be linked into the executable for <core>

Table 3.3

The following table lists the variables that are required to be set by the component's make files, typically "component.mk" file that is found at the top-level directory of the component.

Variables required to be defined by component make files (typically component.mk)		
Sl	**Make variable**	**Description**
1	<mod>_RELPATH	Relative path (from the packages repository of the component) of the module <mod>. Each of the modules that are in the component have to have a separate entry.
2	<mod>_PATH	This has the absolute path of the module <mod>. This has to be set by appending <mod>_RELPATH to <comp>_PATH (<comp> is the name of the component) which would be set in the environment.
3	<mod>_INCLUDE	List of directories (each space separated) that contains the interface header files that are required to be included by other modules and applications that use this module

4	<mod>_BOARD_DEPENDENCY	Specifies if the module <mod> is dependent on boards or not. If set to “yes” (note: all lower case), then the object files and libraries are created under <board> directory, so that each board for which it is built, we’d have a separate copy of the libraries/objs. If it is set to “no” or blank or not defined at all, then it signifies that this module does not change based on boards.
5	<mod>_SOC_DEPENDENCY	Specifies if the module <mod> is dependent on soc or not. If set to “yes” (note: all lower case), then the object files and libraries are created under <soc> directory, so that each soc for which it is built, we’d have a separate copy of the libraries/objs. If it is set to “no” or blank or not defined at all, then it signifies that this module does not change based on soc but in fact depends on board.
6	<mod>_CORE_DEPENDENCY	Specifies if the module <mod> is dependent on core or not. If set to “yes” (note: all lower case), then the object files and libraries are created under <core> directory, so that each core for which it is built, we’d have a separate copy of the libraries/objs. If it is set to “no” or blank or not defined at all, then it signifies that this module does not change based on core.
7	<mod>_APP_STAGE_FILES	List of names of source files (each space separated) belonging to the module <mod> that have to be compiled as a part of the application stage make build step. The file path should be relative to the makefile of the module (ot top-level of the module). This could be link time configuration of the module or any other source that has to be built in the context of an application (typically, if it is app-dependent). If it is blank or not defined at all, then

		it is taken that there are no source files of the module that have to be compiled in the context of the application
8	<mod>_PKG_LIST	List of names of sub-modules that are in the module. Note that each of these sub-modules has to have separate variables described above (1-6). If this is blank or not defined at all, then there are no sub-modules.
9	<comp>_LIB_LIST	List of names of modules that are a part of the component <comp> whose libraries have to be linked when the component is supplied as a pre-built component.
10	<comp>_BOARDLIST	List of BOARD for which this module needs to be built. If board is defined, then it will define SOC based on board. Hence, no need to build define <comp>_SOCLIST Note: Application needs to have <comp>_BOARDLIST defined.
11	<comp>_SOCLIST	List of SOC to which this module needs to be built Note: Module needs to have <comp>_SOCLIST defined.
12	<comp>_<SOC>_CORELIST	List of CORE for a particular SOC to which this module needs to be built

Table 3.4

b. Identifiers

All other identifiers used in the make files are listed in the table below.

Identifiers used in the make files		
Sl	**identifier**	**Description**
1	<board>	`Example: tda2xx-evm idkAM572x evmK2H` Note: These can be extended in future to add similar boards by adding support in platform.mk

2	<SoC>	Example: am572x k2h tda2xx tda3xx For Application/Examples, These values are not directly set by the user. These are derived from the <board>. For Module it is set directly as all the <module> are board independent.
3	<core>	Example: ipu1_0 c66x a15_0 a9_host
4	<ISA>	Example: m4 (ISA value for ARM® Cortex™ M4)

Table 3.5

NOTE: Please note that the values for the identifiers above maybe extended further as we support more boards, SOCs and applications scenarios.

c. *Build options*

The following table lists the build time options that can be either set in build_config.mk or at the command line. The command line settings would take precedence over the ones set in build_config.mk.

Build options that are used in common make files		
Sl	**Make variable**	**Description**
1	ENDIAN	This specifies the endian-ness of the objects and binaries to be built. Valid values are: `big` (big-endian) `little` (little-endian)
2	FORMAT	This specifies the format of the binaries to be built. Valid values are: `COFF` (COFF format) `ELF` (ELF format)
3	BOARD	Board for which the object and binaries are being built for. The value set here would be same that is translated to <board> identifier in the rest of the make files.
4	SOC	soc for which the object and binaries are being built for. The value set here would be same that is translated to <board> identifier for application. For rest of the make files cases where board is not defined, it is needed to be defined.
5	CFLAGS_GLOBAL_<identifier>	C compiler switches that are global in nature (applies to everything that is built) for the given <identifier>. Here <identifier> can be <board>, <core>, <ISA> or <SoC>
6	LNKFLAGS_GLOBAL_<identifier>	Linker switches that are global in nature (similar to CFLAGS)
7	PROFILE_<core>	The profile for which the build is being done for. Valid values: `debug` (Debug profile) `release` (Release profile) The debug profile is typically used

		during development and debugging. However, the performance may not be good. The release profile is typically used after the “debug” build is functionally working. This profile might improve the performance to a great extent, as it optimizes to the highest possible level. Please note the comments in the build_config.mk while setting, as tool-chain for some of the cores (ISAs) don’t support whole_program_debug profile. Also note that more such profiles could be added in the future.

Table 3.6

4. How these make files work?

Now that all the necessary variables and identifiers are familiar, this chapter describes how the common make files work on these variables. This section gives a brief overview of the flow.

a. Building modules

The module level makefile is typically called either while building the application that consumes it or by the release process of the component to which the module belongs.

The make for the module is invoked something like

```
make -C $(<mod>_PATH)
```

which tells make to look out for a "makefile" under <mod>_PATH and build the default target specified. The module makefile sets MODULE_NAME (refer table 3.2) and other required variables in table 3.1. It then "includes" a common make file "common.mk". This file has some of the common make rules and also includes other necessary common make files. The default make target is specified in common.mk, which is to compile and create an archive (library) for the module.

The common make files uses the variables that are set in the module makefile appropriately. For example, the INCLUDE_EXTERNAL_INTERFACES variable is acted upon in the following manner:

A loop iterates for each of the "word"s (component/module name) in this variable and value of <"word">_INCLUDE variable (which has the absolute path of the directories that has the interface header files for the module "word") is used to specify the include search path (for example, prefixing each directory with "-I" switch).

b. Building applications

The application makefile is typically called from a top-level make file. This is required because the same makefile needs to be iterated for each of the cores for which the binaries have to be built. Something like this:

```
make -C $(<app>_EXAMPLE_PATH) CORE=a15_0
make -C $(<app>_EXAMPLE_PATH) CORE=c66x
make -C $(<app>_EXAMPLE_PATH) CORE=ipu1_0
```

Each of the above lines would generate an executable for the core specified. This too, like module makefile, includes "common.mk", which specifies the targets required to build the application. It goes about doing it in the following steps:

- It uses the value of COMP_LIST_<core> to get a list of dependent modules for the application on <core>. It iterates a through a loop for each of these modules and calls the module’s makefile (in the same fashion as described in section 4.a and ensures that the libraries for the module is up-to-date.

- Builds the source files of the application
- Builds those source files of the modules that are marked to be compiled at application stage (via <mod>_APP_STAGE_FILES).
- Links all the module’s libraries and the objects generated in the last two steps and creates an executable

5. Appendix

List of common make files:

Common make files	
common.mk	This is the common make file that defines rules that are common across boards/cores/ISAs. This also includes other common make files
rules_<ISA>.mk	Rules make file for each of the supported ISAs. Example: rules_m3.mk, rules_a8.mk
platform.mk	This make file has all the board specific defines that are commonly used
build_config.mk	This file has build options that are typically changed during the build time
env.mk	This file sets paths for various components, modules and tools that are used within the make files

The last two files – build_config.mk and env.mk are only required to be changed by the end-user who wants to run make to build.